Hazelyn Cates
11/5/22
EN.605.620.81.FA22

Project 2

Implementation and Design

This project implements a total of 14 functions and are described in detail below in sequential order. Functions 1 and 2 implement the hash functions, functions 3 through 8 carry out the insertion of elements into the hash table (note they are identical in function between the division and multiplication schemes), functions 9 and 10 implement an insertion function in the form of chaining when the bucket size of the hash table is three, functions 11 through 13 implement the three collision handling techniques (linear and quadratic probing and chaining) and function 14 prints the resulting hash table, relevant statistics, load factor, and any values that could not be hashed.

This program requires the user to enter the name of the input text file, the bucket size, the hashing scheme of choice, modulo value for division scheme, and multiplication value and constant value for the multiplication scheme.

Hash functions:

division_hashing(number, m)

Takes two arguments, a number from the input data set (number) and the modulo value (m). It performs a division hash function, returning the result of number mod m as the hash value. This function is called in the functions division_insert_linear1, division_insert_quadratic1, division_insert_chaining1, and division_insert_linear3.

multiplication_hashing1(number, c, M)

Takes three arguments: a number from the input data set (number), a constant value greater than 0 and less than 1 (c, entered by the user), and the multiplication value (M). It performs a multiplication hash function, returning the result of $M \times (number \times c \% 1)$ as the hash value. This function is called in the functions multiplication_insertlinear1, multiplication_insert_quadratic1, and multiplication_insert_chaining1.

Insertion into hash table:

division_insert_linear1(hashTable, nums, m, b)

Takes four arguments: an empty hash table with 120 slots (hashTable), the number data from the input file (nums), the modulo value (m), and bucket size of 1 (b). It inserts keys into the hash table by calling the **division_hashing** function and handling any collisions via linear probing.

The function begins by creating an initially empty array called “hash” to keep track of the values calculated by **division_hashing**, as well as initializing variables to store the hash value, the size of the data input, and the number of primary and secondary collisions.

The outer for loop of this function traverses through the data set, having a range of 0 to the size of the data (i.e. how many keys are being hashed) or the size of the hash table, whichever is smaller. It calculates an initial hash value by calling the **division_hashing** function. The first if statement checks if this hash value has already been used and is already present in the “hash” array. If it is not, it gets appended to the “hash” array and the key is added to the hash table at this calculated slot. However, if the hash value is in the “hash” array, indicating the slot in the hash table is already full, then this constitutes a primary collision. In this latter case, a second for loop is entered, whose range is 0 to the size of the hash table (120). The primary collision is resolved by generating a new hash value by calling the **linear_probing** function (see function description below). This new hash value is then checked to see if it has also already been used. If so, it constitutes a secondary collision and the inner for loop continues until a hash value not yet in the “hash” array is calculated.

This repeats for all of the numbers in the input array. Lastly, it calls the **print_hashTable** function (see description of function below).

multiplication_insert_linear1(hashTable, nums, m, c, b)

Identical to **division_insert_linear1** above, this function takes five arguments: an initially empty hash table with 120 slots (hashTable), the number data from the input file (nums), the multiplication value (m), the constant value greater than 0 and less than 1 entered by the user (c), and a bucket size of 1 (b). This function also handles collisions by calling the **linear_probing** function.

The only difference compared to **division_insert_linear1** is that this function calculates its hash values by calling the **multiplication_hashing** function, which is described above. The remaining procedure and output are identical to that described for **division_insert_linear1** and the function ends with a call to **print_hashTable**.

division_insert_quadratic1(hashTable, nums, m, b)

Identical to **division_insert_linear1**, this function takes four arguments: an initially empty hash table of 120 slots (hashTable), the numerical data from the input file (nums), the modulo value (m), and a bucket size of 1 (b). The **division_hahsing** function is called to generate the hash values.

The only difference compared to **division_insert_linear1** is that instead of handling collisions using the **linear_probing** function, this function uses the **quadratic_probing** function, which is described below. The remaining methodology is identical to the **division_insert_linear1** function.

multiplication_insert_quadratic1(hashTable, nums, m, c, b)

Identical to **division_insert_quadratic1**, this function takes five arguments: an initially empty hash table of 120 slots (hashTable), the numerical data from the input file (nums), the multiplication value (m), the constant value greater than 0 and less than 1 entered by the user (c), and a bucket size of 1 (b). This function calls the **quadratic_probing** function (described below) to handle collisions.

The only difference compared to **division_insert_quadratic1** is that this function calls the **multiplication_hashing** function to generate the hash values. The remaining methodology is identical to the **division_insert_quadratic1** function.

division_insert_chaining1(hashTable, nums, m, b)

Takes four arguments: an initially empty hash table of 120 slots (hashTable), the numerical data from the input file (nums), the modulo value (m), and a bucket size of 1 (b). It inserts keys into the hash table by calling the **chaining** function (described below), which functions as the collision handling method.

In this function, an initially empty array called “count” is initialized to keep track of the number of primary collisions. In this case, there are no secondary collisions since multiple values can be hashed to the same slot.

There is only one for loop that has a range of 0 to the size of input file or the size of the hash table, whichever is smaller. The hash value is calculated by calling the **division_hashing** function (see above for description) and then the **chaining** function is called to append the key to the slot calculated by **division_hashing**. Every time a key is hashed to a slot in the hash table, the count array is incremented by one that that index, and a primary collision is recorded if there is a value greater than 1 at that index in the count array. Lastly, the **print_hashTable** function is called, which is described below.

multiplication_insert_chaining1(hashTable, nums, m, c, b)

Identical to **division_insert_chaining1**, takes five arguments: an initially empty hash table with 120 slots (hashTable), the numerical data from the input file (nums), the multiplication value (m), the constant value greater than 0 and less than 1 entered by the user (c), and a bucket size of 1 (b).

The only difference compared to the **division_insert_chaining1** function is that this function calculates the hash values by calling the **multiplication_hashing** function. The remaining methodology is identical to **division_insert_chaining1**.

division_insert_linear3(hashTable, nums, m, b)

Takes four arguments: an initially empty hash table with 40 slots (hashTable), the numerical data from the input file (nums), the modulo value (m), and a bucket size of 3 per slot (b), for a total of 120 spaces.

Similar to the **division_insert_chaining1** and **multiplication_insert_chaining1** functions, this function utilizes an initially empty array called "count", which keeps track of the number of primary collisions.

The outer for loop of this function traverses through the input data, having a range from 0 to the size of the data or to the size of the hash table, which is less. It calculates a hash value by calling the **division_hashing** function. After calculating this value, it performs an error check to see if the calculated hash value exceeds the size of the hash table. If so, the **linear_probing** function is called to calculate a new hash value, and this continues until an appropriate hash value is calculated.

Since this function is called when bucket size = 3, only three values per slot can be hashed to the hash table. Every time a value is hashed to the same slot, the corresponding index in the "count" array is incremented by 1. If there are less than three values in a given slot, the **chaining** function is called to append the key to that slot in the hash table and the corresponding index in "count" is incremented by 1.

If the "count" array reaches three for a given index, then no more values can be hashed to that index in the hash table and a primary collision occurs. If this is the case, a second, inner for loop is entered, which has the range 0 to 41 (which is one more than the size of the hash table), and the **linear_probing** function is called to generate a new hash value. Once again, this hash value is checked to see if it exceeds the size of the hash table and if so, the **linear_probing** function is called to generate hash values until an appropriate one is reached. Once an acceptable hash value is calculated, if this new location in the hash table has a value less than three in its corresponding index in the "count" array, then the value can be hashed to that index. However, if the new hash value also corresponds to a full slot, then a secondary collision is recorded and the inner for loop continues. Lastly, this function calls the **print_hashTable** function.

division_insert_quadratic3(hashTable, nums, m, b)

Identical to **division_insert_linear3** and takes four arguments: an initially empty hash table of size 40 (hashTable), the numerical data from the input file (nums), the modulo value (m), and a bucket size of 3 (b).

The only difference compared to the **division_insert_linear3** function is that instead of linear probing, this function calls the **quadratic_probing** function if a collision occurs to calculate a new hash value of if the hash value exceeds the size of the hash table, as described above. The remaining methodology is identical to **division_insert_linear3**.

Collision handling:

linear_probing(hash_value, m, i)

Takes three arguments: the hash value initially calculated in any of the insert functions described above (hash_value), the modulo or multiplication value (m), and i, the current index of the key in the data set. This function returns (hash_value + i) % m, which serves as the new hash value in response to a collision.

quadratic_probing(hash_value, m, i)

Takes three arguments: the hash value initially calculated in any of the insert functions described above (hash_value), the modulo or multiplication value (m), and i, the current index of the key in the data set. This function returns (hash_value + (0.5 × i) + (0.5 × (i^2))) % m, which serves as the new hash value in response to a collision.

chaining(hashTable, hash_value, num)

Takes three arguments: the hash table being used (hashTable), the current hash value calculated by a call to **division_hashing** (hash_value), and the current number in the data set (num).

This function appends the current number in the data set to its corresponding slot determined by the hash value in the hash table.

If the bucket size is set to 1, there is no restriction on how many values can be hashed to the same slot, but the number of values that can be hashed is still restricted by the size of the hash table. If the bucket size is 3, only three values can be hashed to a single slot, and this restriction is handled in the **division_insert_linear3** and **division_insert_quadratic3** functions.

Printing the hash table:

print_hashTable(hashTable, nums, b, p, s, m, scheme, col, total)

Takes nine arguments: the hash table (hashTable), the numerical data from the file input (nums), the bucket size (b), the number of primary collisions from the respective insert function (p), the number of secondary collisions from the respective insert function (s), the mod or multiplication value (m), the scheme used (scheme), the collision handling technique used in the respective insert function (col), and the total number of values able to be hashed (total).

This function is called in every insert function (see above for their descriptions) and prints out the size of input data, the bucket size, the hashing scheme, the collision handling method, the number of primary and secondary collisions, the resulting hash table, the load factor, and if applicable, any values unable to be hashed. All of the results from each hashing function are appended to the end of the same output file, and each run is separated by asterisks and the statement “Next Run”.

Hazelyn Cates
11/5/22
EN.605.620.81.FA22

Efficiency and Analysis

There are various different hashing and collision handling techniques that can be implemented with hashing values to a hash table, each with its own benefits and shortcomings. In this project, only division and multiplication hash functions are utilized:

Hashing by division is a simple and fast method, defined by the following hash function:

$$h(k) = k \bmod m$$

Where k is the key in the input being hashed and m is the hash table size (pg. 263, Cormen et al, 2009). The result of this modulo function gives the slot in the hash table to which the key k is hashed. The benefits of hashing by division are dependent on the value of m; prime numbers and numbers that are not a power of 2 can help prevent large amounts of collisions from occurring. (pg. 263, Cormen et al, 2009).

Next, hashing by multiplication is defined by the following hash function:

$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$

Where m is the size of the hash table, k is the key to be hashed, and A is constant value between 0 and 1, exclusive (pg. 263, Cormen et al, 2009). It is noteworthy that using multiplication, the value of m is not as influential to the result as it is in hashing by division, due to the fact that the value of m is being multiplied, not divided, and therefore uniformly impacting the result (pg. 264, Cormen et al, 2009).

Once a value is hashed by either of the two methods described above, a collision of keys to the same slot is inevitable as input size increases. There are multiple collision handling methods in hashing, and the three utilized in this program are linear probing, quadratic probing, and chaining via an open addressing scheme. The methods are described as follows:

Linear probing uses a sequential search pattern to find the next open slot in the hash table by incrementing the hash value first by one, then by two and so on until, if necessary, the entire hash table has been probed (“Open Addressing”, 2022). While easy to understand and convenient in implementation, a downside is that primary clustering of values can take place, in which values get hashed in clusters within the table (i.e. in sequential slots) instead of being distributed in a more uniform fashion around the table (pg. 272, Cormen et al, 2009). This can increase the time it takes to probe for an empty spot, thus negatively impacting run time (pg. 272, Cormen et al, 2009).

Similar to linear probing, quadratic probing searches for the next empty slot in the hash table quadratically using restricted constant values, thus decreasing the chance of hashing values in primary clusters in the hash table, mentioned above being an issue in linear probing (pg. 272, Cormen et al, 2009). While able to distribute values more uniformly around the hash table, quadratic probing can still result in secondary clustering, in which two values being hashed have the same initially calculated probe position, indicating that they have the same probe sequence (pg. 272, Cormen et al, 2009). In both linear and quadratic probing, the number of distinct probe sequences (which are lists of slots that get calculated

when a collision occurs in hashing a value) are dependent on the size of the hash table (pg. 272, Cormen et al, 2009).

Different from both linear probing and quadratic probing is chaining. When a collision occurs when hashing a key, instead of calculating a new hash value, chaining adds the value to a linked list in the same slot ("Implementation", 2022). When using chaining, the number of values that can be hashed is still limited to the size of the hash table and the space available, but not limited in how many values can be hashed to the same slot, as appending in Python takes O(1) time (Mahajan, 2022).

In the case of any of the collision handling methods described above, if the size of the input data exceeds the size of the hash table, then all values in the input will not be able to be hashed and the hash table will have a load factor greater than 1. When this occurs, the hash table must be resized (i.e. rehashed), in which its size is doubled to accommodate all of the values (Sehgal, 2022). While rehashing was not implemented in this program, it is important to keep the load factor at an acceptable value.

To elaborate, the load factor of a hash table is defined as the quotient of the number of values hashed divided by the size of the hash table (Sehgal, 2022). Rehashing of a table can be set to occur if the load factor hits a certain threshold value, with a common default load factor value being 0.75 ("Load Factor", 2022). The benefit of rehashing once this certain load factor value is hit is that it will reduce the number of collisions that take place. Theoretically, in the best case, calculating the hash value, inserting a value and deleting a value from a hash table take O(1) time ("Time and Space,", n.d.). Also theoretically, in the worst case, inserting a value and deleting a value from the hash table could take O(n) time due to the number of collisions and the need to utilize linear probing, quadratic probing, or chaining many times, and if the value being deleted is located in the last slot of the table (respectively) ("Time and Space", n.d.).

Additionally, the time complexity of each function and therefore the time complexity of each function call are dependent on the size of the input file (i.e. how many values are being hashed). As the size of the input file increases, so will the number of collisions. As the hash table fills up and collisions become more frequent, if the table is rehashed more calls to the collision handling functions will be required, and in the worst case for this project and its given parameters, up to 120 calls of each collision handling method per scheme will be required to hash all values to the table, namely if the size of input file is the same size as the hash table (i.e. there are as many numbers to be hashed as there are slots in the hash table) or the number of input values exceeds the size of the hash table. In both cases, the hash table will be completely full.

Deleting a value from a hash table, as mentioned above, takes O(1) in the best case and O(n) in the worst case. In the best case, deleting a value from a hash table would involve a search in the hash table where the resulting key is the first value or very close to the top of the hash table, with minimal traversals through the hash table needed. By comparison, if the value to be deleted is towards the bottom of the hash table, then the majority or entire hash table would have to be searched first in order to find the key to be deleted, taking up to n searches (hence, O(n)) ("Time and Space", n.d.). Once a value is deleted, that slot in hash table is empty again for a new value to be hashed. However, since hash tables have O(n) space complexity, where n is the number of slots, this could potentially cause a problem when deleting values, especially if the hash table was rehashed. If enough values are deleted from a hash table

that was rehashed that the extra space is no longer required, the hash table then takes up excessive and unnecessary space, which could pose an issue if there is limited space in memory.

When looking at the space complexity of hashing using the linear probing, quadratic probing, or chaining collision handling technique, it will always require O(n) space since each value in the hash table is stored in the memory, therefore n is the number of keys hashed ("Time and Space", n.d.).

Results

To test this program, six file inputs were utilized, whose number of keys included 36, 60, 84, 104, 120 and 150. As was observed, the number of collisions that occur for a given data set are directly dependent on the size of the data set. For small data sizes, collisions are less likely to occur due to the greater likelihood that the hash values calculated are going to be unique. Additionally, for small sized datasets, the resulting hash table for each hashing scheme will be very similar since very few collisions occur and therefore new hash values will not have to be calculated. However, as the size of the data increases, collisions become more and more frequent. It is also important to note that the numbers in the input file directly affect if they will they be able to be hashed or not. For example, if the values in the input are all the same, then they will all be hashed to the same slot in the table. Or if the values in the input are not very compatible with the mod value in a division hash function, then it is possible that some values will not be hashed.

The following tables represent the run times for each hashing scheme and the corresponding collision handling methods for the six datasets of size: 36, 60, 84, 104, 120, and 150:

Dataset size = 36

Hashing Scheme	Bucket Size	Collision Handling	Observed Run Time (sec)
Division modulo 120	1	Linear probing	1.66e-03
Division modulo 120	1	Quadratic probing	1.33e-03
Division modulo 120	1	Chaining	1.25e-03
Division modulo 113	1	Linear probing	1.95e-03
Division modulo 113	1	Quadratic probing	1.26e-03
Division modulo 113	1	Chaining	1.42e-03
Division modulo 41	3	Linear probing	1.93e-03
Division modulo 41	3	Quadratic probing	1.12e-03
Multiplication 120	1	Linear probing	1.73e-03
Multiplication 120	1	Quadratic probing	1.62e-03
Multiplication 120	1	Chaining	1.28e-03

Table 1: observed run times for a dataset size of 36.

Dataset size = 60

Hashing Scheme	Bucket Size	Collision Handling	Observed Run Time (sec)
Division modulo 120	1	Linear probing	1.85e-03
Division modulo 120	1	Quadratic probing	1.30e-03
Division modulo 120	1	Chaining	1.62e-03
Division modulo 113	1	Linear probing	1.57e-03
Division modulo 113	1	Quadratic probing	1.35e-03
Division modulo 113	1	Chaining	1.00e-03
Division modulo 41	3	Linear probing	1.46e-03
Division modulo 41	3	Quadratic probing	1.14e-03
Multiplication 120	1	Linear probing	1.63e-03
Multiplication 120	1	Quadratic probing	1.68e-03
Multiplication 120	1	Chaining	9.75e-04

Table 2: observed run times for a dataset size of 60.

Dataset size = 84

Hashing Scheme	Bucket Size	Collision Handling	Observed Run Time (sec)
Division modulo 120	1	Linear probing	1.93e-03
Division modulo 120	1	Quadratic probing	1.53e-03
Division modulo 120	1	Chaining	1.33e-03
Division modulo 113	1	Linear probing	1.95e-03
Division modulo 113	1	Quadratic probing	2.14e-03
Division modulo 113	1	Chaining	1.43e-03
Division modulo 41	3	Linear probing	1.91e-03
Division modulo 41	3	Quadratic probing	1.23e-03
Multiplication 120	1	Linear probing	1.77e-03
Multiplication 120	1	Quadratic probing	1.70e-03
Multiplication 120	1	Chaining	9.77e-04

Table 3: observed run times for a dataset size of 84.

Dataset size = 104:

Hashing Scheme	Bucket Size	Collision Handling	Observed Run Time (sec)
Division modulo 120	1	Linear probing	2.65e-03
Division modulo 120	1	Quadratic probing	1.45e-03
Division modulo 120	1	Chaining	1.84e-03
Division modulo 113	1	Linear probing	1.81e-03

Division modulo 113	1	Quadratic probing	1.29e-03
Division modulo 113	1	Chaining	8.63e-04
Division modulo 41	3	Linear probing	2.13e-03
Division modulo 41	3	Quadratic probing	1.48e-03
Multiplication 120	1	Linear probing	2.77e-03
Multiplication 120	1	Quadratic probing	5.25e-03
Multiplication 120	1	Chaining	8.45e-04

Table 4: observed run times for a dataset size of 104.

Dataset size = 120:

Hashing Scheme	Bucket Size	Collision Handling	Observed Run Time (sec)
Division modulo 120	1	Linear probing	4.49e-03
Division modulo 120	1	Quadratic probing	2.47e-03
Division modulo 120	1	Chaining	1.12e-03
Division modulo 113	1	Linear probing	6.10e-03
Division modulo 113	1	Quadratic probing	3.75e-03
Division modulo 113	1	Chaining	8.03e-04
Division modulo 41	3	Linear probing	1.57e-03
Division modulo 41	3	Quadratic probing	1.92e-03
Multiplication 120	1	Linear probing	3.57e-03
Multiplication 120	1	Quadratic probing	3.10e-03
Multiplication 120	1	Chaining	8.55e-04

Table 4: observed run times for a dataset size of 120.

Dataset size = 150:

Hashing Scheme	Bucket Size	Collision Handling	Observed Run Time (sec)
Division modulo 120	1	Linear probing	3.55e-03
Division modulo 120	1	Quadratic probing	2.11e-03
Division modulo 120	1	Chaining	9.01e-04
Division modulo 113	1	Linear probing	3.67e-03
Division modulo 113	1	Quadratic probing	3.24e-03
Division modulo 113	1	Chaining	9.88e-04
Division modulo 41	3	Linear probing	1.71e-03
Division modulo 41	3	Quadratic probing	1.51e-03
Multiplication 120	1	Linear probing	5.03e-03
Multiplication 120	1	Quadratic probing	1.18e-02

Multiplication 120	1	Chaining	1.81e-03

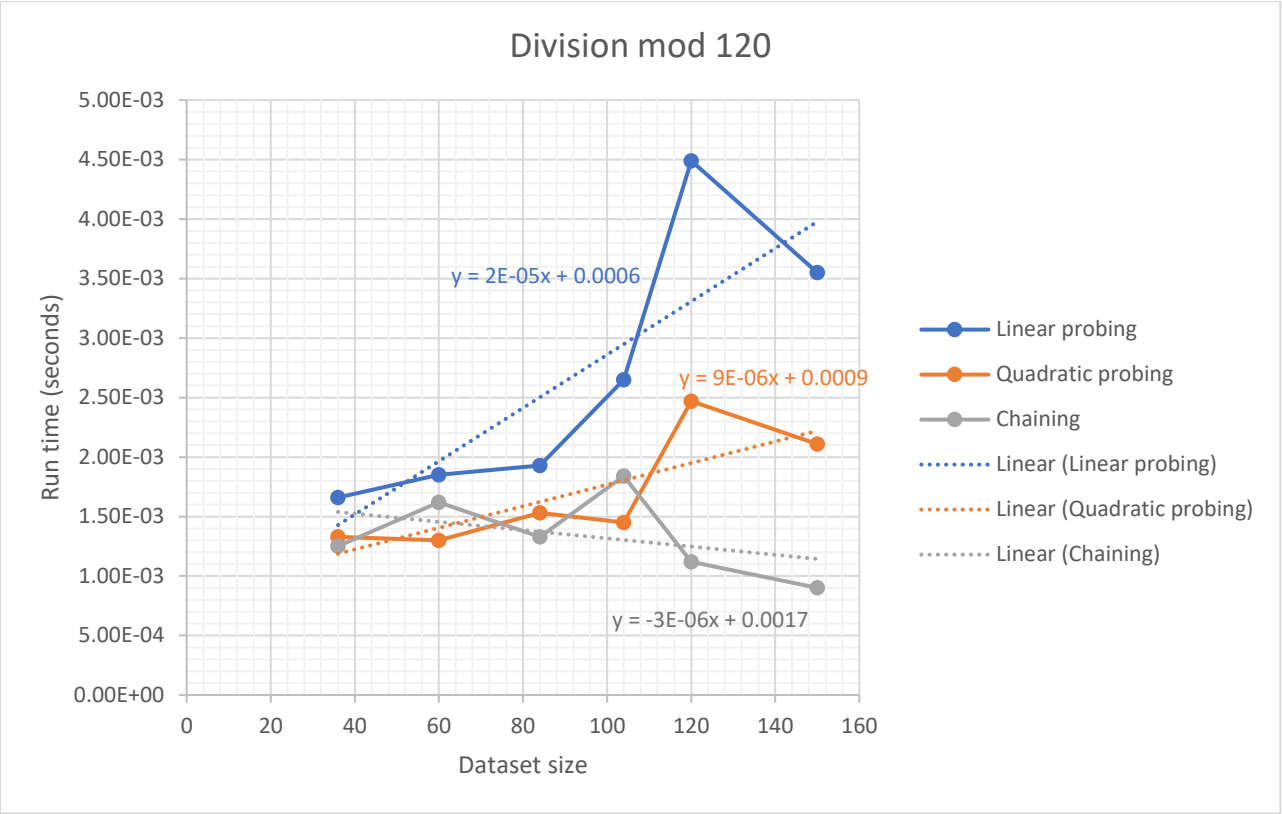
Table 5: observed run times for a dataset size of 150.

As can be observed in the above tables, the run times for all schemes are very similar for each of the six dataset sizes. This was a bit surprising, as it would be expected that as dataset size increased and the number of required collision handling calls also increased, that the run time would reflect this. Calculating a new hash value via linear probing, quadratic probing and chaining all take constant time (O(1)) in the best case and linear time (O(n)) in the worst case, the latter requiring probing all n slots in the table ("Hash Table", n.d.). It also should be noted that the task appending the results of each run of the program to the end of the same file takes constant time and did not impact the run time as the runs were done sequentially ("Time and Space", n.d.).

From the above tables, the chaining collision handling methods for hash schemes 1, 2 and 4 ran slightly faster than the linear or quadratic probing schemes in all six datasets. This is likely attributed to the fact that chaining does not require a new hash value to be calculated and as mentioned above, appending takes O(1) time. For the six datasets, using a hash table with a bucket size of 3 versus 1 did not influence run time, since both tables still have 120 total slots.

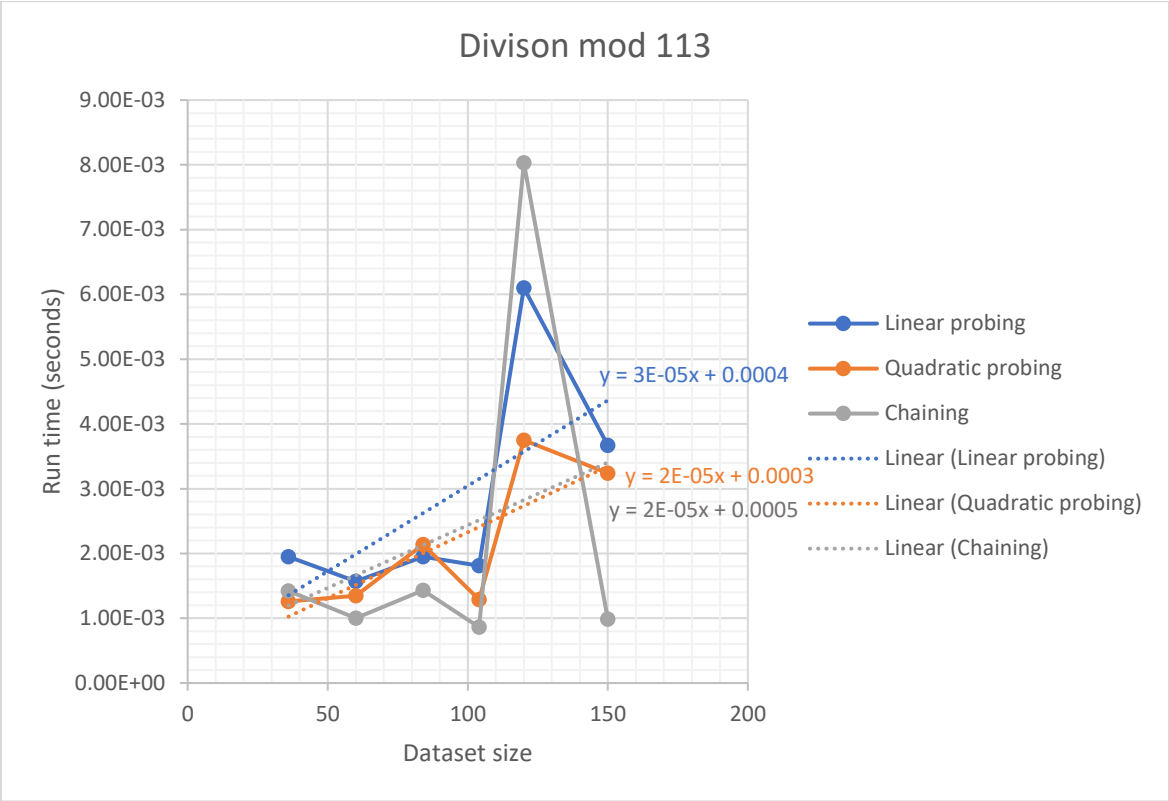
The run time of each scheme for each dataset size was plotted and were grouped by hash scheme to produce the following four graphs:

Hazelyn Cates
11/5/22
EN.605.620.81.FA22



Graph 1: run times of each collision handling method using the hashing scheme division mod 120 on dataset sizes 36, 60, 84, 104, 120, and 150 with a bucket size equal to 1. Note that equations seen on the graph are color-coded and correspond to each respective trendline (dotted line).

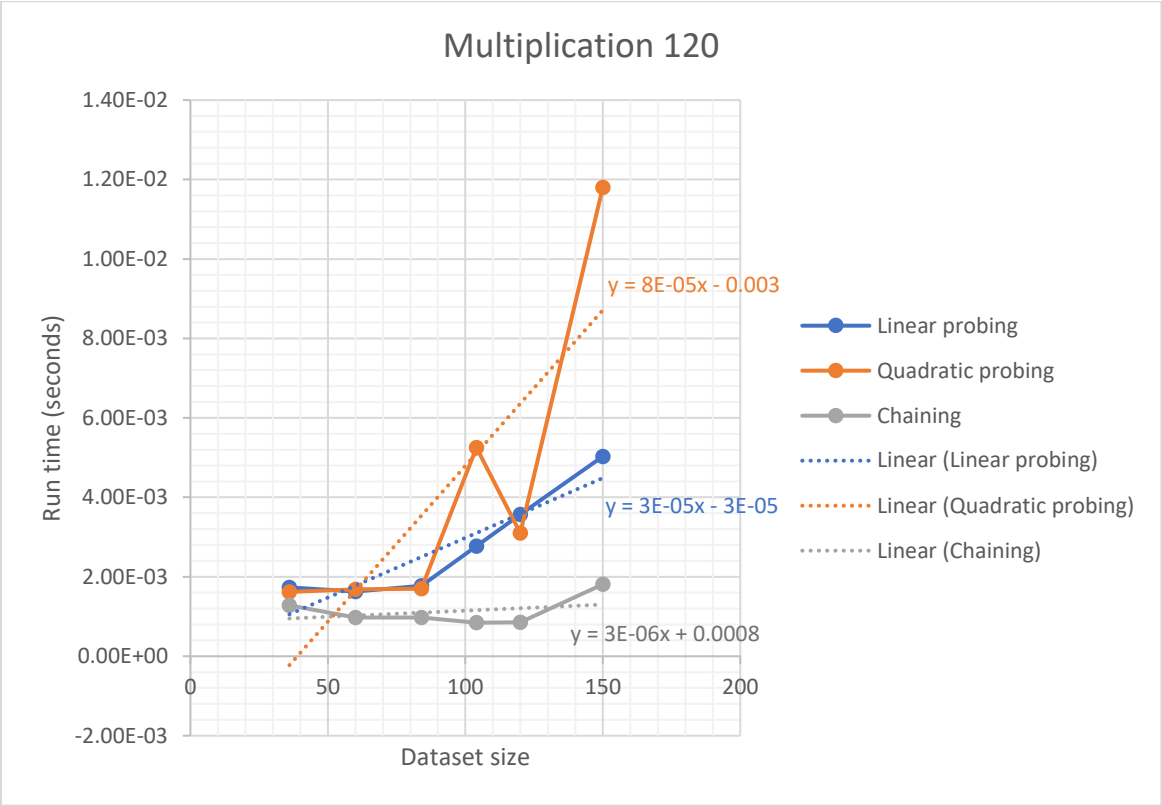
Hazelyn Cates
11/5/22
EN.605.620.81.FA22



Graph 2: run times of each collision handling method using the hashing scheme division mod 113 on dataset sizes 36, 60, 84, 104, 120, and 150 with a bucket size equal to 1. Note that equations seen on the graph are color-coded and correspond to each respective trendline (dotted line).

Graph 3: run times of each collision handling method using the hashing scheme division mod 41 on dataset sizes 36, 60, 84, 104, 120 and 150 with a bucket size equal to 3. Note that equations seen on the graph are color-coded and correspond to each respective trendline (dotted line).

Hazelyn Cates
11/5/22
EN.605.620.81.FA22



Graph 4: run times of each collision handling method using the hashing scheme multiplication value 120 on dataset sizes 36, 60, 84, 104, 120 and 150 with a bucket size equal to 1. Note that equations seen on the graph are color-coded and correspond to each respective trendline (dotted line).

As can be seen in the above graphs, the run times are variable and observably erratic in some of the hashing schemes. In the smaller datasets (i.e. 36, 60, 84, 104), the theoretical run times of linear probing, quadratic probing, and chaining, as stated above, is O(1) in the best case and O(n) in the worst case.

In graph 1, which shows the run times of the division mod 120 hash scheme with a bucket size of 1 for all dataset sizes (x-axis), the linear and quadratic probing lines (blue and orange, respectively) follow an upward trend up to dataset size 120. This is expected since as the number of values being hashed increases, so does the number of collisions and thus the required calls to the respective collision handling functions. However, as observed in graph 1, there is a decrease in run time for a dataset size of 150. It would be expected that the run time would increase linearly along with dataset size. The line representing the chaining method (gray) follows a similar trend, with an increase in run time linear to dataset size for datasets 36, 60, 84 and 104. However, a decrease in run time is observed for data sets of size 120 and 150. Once again, this was not expected, as it was expected that as dataset size increases, so would the run time in a linear manner. The trendlines for all three collision handling schemes in graph 1 are linear in the form of y = mx + b. It can be observed that the line equation for chaining has a negative slope value and a decreasing trendline, indicating an overall decreasing trend in run time. The other two

trendlines associated with linear probing and quadratic probing have positive slops and an overall increasing trend in run time.

In graph 2, which shows the results of the division mod 113 hash scheme for all dataset sizes with a bucket size of 1, all three hashing schemes follow almost the same trends. As can be seen in the graph, there is a decrease in run time between datasets of size 36 and 60, an increase in run time between datasets of size 60 and 84, a decrease between datasets of size 84 and 104, a sharp increase from dataset sizes 104 and 120, and a sharp decrease in runtime between datasets of size 120 and 150. It does make sense that the run time would be high with a dataset size of 120, since all slots in the hash table would be full and the collision handling schemes would run in the worst case, being called n times since rehashing does not occur. However, an increase in run time proportional to dataset size was expected, and like in graph 1, there is a decrease in run time between datasets of size 120 and 150, while the expected would be a continued increase and a peak run time for the dataset of size 150. Also, for each collision handling scheme, the respective trendlines and their associated equations are shown. All three trendline equations are linear and have a positive slope, indicating an overall increasing trend in run time with respect to dataset size.

For graph 3, showing the results of the division mod 41 hash scheme for all dataset sizes with a bucket size of 3, the run times for linear probing (blue line) are erratic and follow no observable trend. The datasets of size 60, 120 and 150 have the smallest run times for linear probing and the data set of size 104 has the largest run time for linear probing. By comparison, the run times for quadratic probing (orange line) follow a more linear increasing trend for datasets of size 36, 60, 84, 104, and 120. However, like seen in graphs 1 and 2, there is a decrease in run time for the dataset of size 150, which was unexpected. Also, for both the linear and quadratic run times, the trendlines for each can be seen. The linear probing trendline has a negative slope, indicating an overall decreasing trend in run time with respect to dataset size. For quadratic probing, its trendline has a positive slop, indicating an overall increasing trend in run time with respect to dataset size. The reason for the dramatic differences in run time for the different datasets compared to the theoretical run time is unknown.

Lastly, for graph 4, which shows the results of the multiplication hashing scheme with a value of 120 and a bucket size of 1, the run times for all datasets follow a mostly linear increasing trend, namely the linear probing (blue) line. The quadratic probing line (orange) shows a sharp increase in run time for a dataset size of 84 to 104, a decrease from datasets 104 to 120, and then a sharp increase in run time between data sets of size 120 and 150. Interestingly, it can be observed that the run time using the chaining method (gray line) follows a somewhat constant trend for datasets of size 36, 60, 84, 104, and 120 before increasing slightly for a dataset of size 150. This line most closely represents the most ideal and expected results of running in near constant time for the majority of dataset sizes. The trendlines associated with all three lines in this graph have positive slope values, indicating an overall increasing trend in run time respective to dataset size.

It is important to note that the input values used in this program were randomly generated, and the values in the input can affect run time positively or negatively, depending on if they are powers of 2, prime, even, odd, etc. This in turn can dictate how many collisions occur and therefore how many times

the collision handling methods need to be called, which can potentially make a program go from running in constant time to running in linear time with respect to input size.

Additionally, the constant value chosen by the user for the multiplication scheme does not seem to affect run time since it is just a constant value. However, it was observed to influence the calculated hash values. It was observed that if the value of the constant was changed between running the same input file, the resulting hash table changed, in some cases hashing all the values while in others being unable to hash up to 40 values. Its influence is also notably seen in the chaining collision handling function where depending on the constant value, the numbers are chained together in a select few slots instead of being more dispersed around the table. In the multiplication hash function, the multiplication value M stays the same, but as the constant value c changes, it influences the result of the multiplication with the value in the input file and the resulting mod, which is always 1.

In order to account for any values that could not be hashed, the appropriate error handling methods were put in place to account for these values. In almost all cases, the values that couldn't be hashed were entirely dependent on the dataset and their compatibility with the mod value. The number of values hashed were tracked in each insertion function using an array. In the **print_hashTable** function, the number of values that could not be hashed were calculated by subtracting the length of the hash table by the total number of values hashed. Additionally, an initial issue encountered when using a bucket size of 3 was that hash values greater than 39 (since indexing of the hash table starts at 0) were being calculated. Since with a bucket size of 3 the hash table is limited to 40 (i.e. 39 when indexing from 0) addressable slots, error handling had to be put in place to detect any of these hash values. If a hash value over 39 was detected, the respective probing function was called again to calculate a new hash value.

Given the results in the above tables and graphs, it is difficult to discern which hashing method and which collision handling method is superior above the next, but from the graphs described above, the multiplication hash scheme with a multiplication value of 120 gave the best results when using the chaining method. However, it should be noted that the division mod 113 scheme performed well up to a size of 120, where again the reason for the sudden decrease in run time for an input size of 150 is not entirely known.

Bioinformatics Application

Hashing is a very useful tool for data collection, data querying, and data access, providing an efficient means of storing many different types of data, including that of DNA and RNA sequence data, genome assembly data, and sequence alignment data (Mohamadi et al, 2016).

A paper from 2016 by Mohamadi et al describes a hashing algorithm called "ntHash" whose use is processing DNA and RNA sequencing data, namely for calculating the hash values of sequential k-mers. K-mers are substrings of a DNA sequence of length *k* and the counting of them plays an important role in genome sequencing, genome and transcriptome assembly, and error analysis of sequence reads (Melsted & Pritchard, 2011). The algorithm presented by Mohamadi et al is recursive, where the hash value of the next k-mer is obtained from the hash value of the one before it (2016). More specifically,

this paper described the ntHash algorithm as utilizing cyclic polynomial hashing, which is a type of rolling hash function (i.e. the hash function utilizes addition but not multiplication) to generate the hash values (Mohamadi et al, 2016; “String hashing”, 2022). It was stated by Mohamadi et al that the ntHash algorithm runs in O(l + k) time, compared to O(lk) time of conventional hash functions (where “l” is length and “k” is the k-mer) (2016). Additionally, this paper described various test cases with k-mers ranging in length from 50 to 250 nucleotides where ntHash outperformed other k-mer hashing algorithms (CityHash, MurmurHash, xxHash, and ntBase) by running up to 20 times faster than the competing algorithms (Mohamadi et al, 2016).

Another example of hashing’s use in bioinformatics is its application in a paper from 2016 by Wu, which describes the benefit of hashing and data compression of genome sequence data to increase the speed of genome alignment (Wu, 2016). A hash table can represent the sequence of a genome as lists of genomic positions for each k-mer or oligomer, and hash tables are used in this manner in various programs including BLAT and GMAP, which are both genome alignment tools (Wu, 2016). When aligning genomic sequences, hash tables can aid in aligning sequences that contain mismatches or single nucleotides polymorphisms (SNPs), since the potentially differing k-mers have the same corresponding genomic position (Wu, 2016).

To continue, in the paper, Wu addresses the limitations of a lookup hash table, in which the hash table contains a list of k-mer positions, by describing a form of data compression called bitpacking (2016). When using a lookup hash table to store genomic information, an offset array, which stores pointer information of the current k-mer position and the proceeding k-mer position, grows exponentially in size, directly proportional with the size of k (Wu, 2016). By compressing the data stored in the offset array, more data can be stored in less memory (Wu, 2016). By bitpacking and vectorizing the offset array data (which is sorted in monotonically increasing order), that is, using the fewest bits possible to store each integer and process them in parallel (respectively), Wu introduced the data structures and algorithms BP32-columnar and BP64-columnar (2016). By compressing the data in the hash table using these two schemes, Wu found that they both resulted in increased retrieval speed of data for genomic alignment, with BP64-columnar having a more balanced time-to-space tradeoff (2016).

Similar in idea to the article by Wu, a paper from 2018 by Girotto et al describes a hashing approach of DNA sequences by using spaced-seed hashing, which is essentially a pattern for string matching that uses 1’s for absolute matches and 0’s for positions that are adaptable (i.e. can tolerate mismatches), and are used in sequence alignment. In order for a DNA k-mer to be hashed, it must first be converted into a binary string, in which the spaced seed can now be used to find matches within that DNA k-mer, as shown in figure 1 from the paper by Girotto et al (2018):

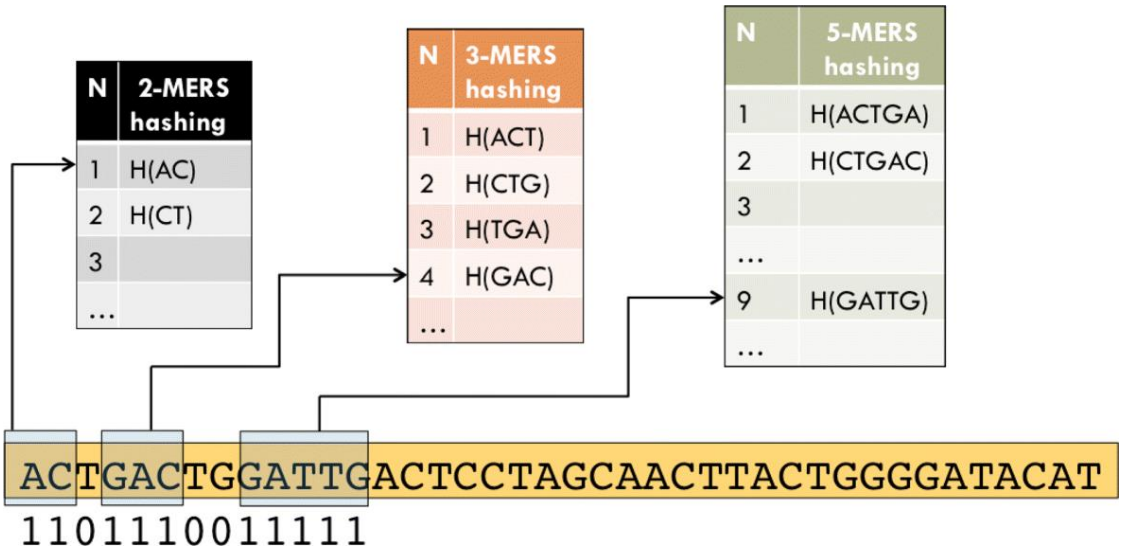


Figure 1 (Girotto et al, 2018)

Girotto et al describes the Fast Indexing for Spaced seed Hashing (FISH) algorithm, which was developed to increase the speed at which hash values of spaced seeds are computed, while at the same time maximizing the probability of a match between the spaced seed and the DNA k-mer and maximizing the sensitivity of the search (Girotto et al, 2018). Girotto et al found that depending on the length of the input, FISH can increase this speed between 1.9 and 6.03 times faster than traditional methods and that for longer reads, the speed of FISH increases, having implications for bioinformatic tools like BLAST, which use k-mers for local alignment of sequences (2018).

Reflection

This project was very challenging and required a lot of planning and a solid understanding of hashing before implementation into code.

If I were to redo this project, I would attempt to consolidate the functions into fewer functions to improve readability and potentially efficiency of the code. For example, in this program, the division and multiplication insert schemes are written separately even though they carry out identical methods. While combining them would not improve efficiency, it would improve the readability of the code. Related to that, when the bucket size is set to 1 by the user, each insert function for the division and multiplication schemes carry out the three collision methods separately, although since each collision method was written as a separate function, the could in theory all be included in one function and called at the same time. Also, the linear probing and quadratic probing methods were implemented in separate functions, but they too could also have been consolidated in one.

Additionally, the output of this program was written to a single file. Each time the program is run, if the file is not deleted, the results are appended to the end of the file. This can make readability difficult, especially if the program is run many times without deleting the file, where the file gets fairly

long, and the newest results are at the bottom. To fix this, multiple output files could be used, but that has the potential of becoming convoluted and confusing for the user.

The design choice described in the first section of this report was for the benefit of the programmer, that is, to organize the functions individually to ensure all requirements were included and implemented properly. Consolidation of the functions, as described above, was considered but ultimately not taken into effect in the final code to prevent any potential accidental loss of functionality.

Entering this project, I was already familiar with defining functions and calling them elsewhere, and this project also provided another opportunity to use regex and the time library This project allowed me to become much more familiar and comfortable in dealing with arrays, function calls, function arguments, and file input and output. Ultimately, while this project was very difficult and took a lot of trial and error, it improved my programming skills and my critical thinking skills on how to identify and solve a problem(s) successfully.

And just by the nature of this project, I now have a much greater appreciation and understanding of hashing and its uses both in general and in a bioinformatics context.

Hazelyn Cates
11/5/22
EN.605.620.81.FA22

References:

Hash Table and Hash Map. (n.d.). Stepik. https://stepik.org/lesson/31445/step/7

Load Factor and Rehashing. (2022). GeeksforGeeks. https://www.geeksforgeeks.org/load-factor-and-rehashing/

Mahajan, Rahul. (2022). Append in Python. *Scaler Topics*. https://www.scaler.com/topics/append-in-python/

Melsted, Páll & Jonathan K Pritchard. (2011). Efficient counting of k-mers in DNA sequences using a bloom filter. *BMC Bioinformatics, 12*(333). https://doi.org/10.1186/1471-2105-12-333

Mohamadi, Hamid et al. (2016). ntHash: recursive nucleotide hashing. *Bioinformatics*, *32*(22), 3492-3494. 10.1093/bioinformatics/btw397

Open Addressing Collision Handling technique in Hashing. (2022). GeeksforGeeks. https://www.geeksforgeeks.org/open-addressing-collision-handling-technique-in-hashing/

Samuele Girotto et al. (2018). Efficient computation of spaced seed hashing with block indexing. *BMC Bioinformatics, 19*(Suppl 15), 441. https://doi.org/10.1186/s12859-018-2415-8

Sehgal, Anmol. (2022). Load Factor and Rehashing. *Scaler Topics*. https://www.scaler.com/topics/data-structures/load-factor-and-rehashing/

String hashing using Polynomial rolling hash function. (2022). GeeksforGeeks. https://www.geeksforgeeks.org/string-hashing-using-polynomial-rolling-hash-function/

Time and Space Complexity of Hash Table operations. (n.d.). OpenGenus. https://iq.opengenus.org/time-complexity-of-hash-table/#:~:text=The%20hash%20key%20is%20calculated,complexity%20is%20O(1)

Wu, Thomas D. (2016). Bitpacking techniques for indexing genomes: I. Hash tables. *Algorithms for Molecular Biology, 11*(5). https://doi.org/10.1186/s13015-016-0069-5

Resources used for writing the program:

Cormen, H. Thomas et al. (2009). Introduction to Algorithms. 3rd ed. The MIT Press.

Hash Functions and list/types of Hash functions. (2022). GeeksforGeeks. https://www.geeksforgeeks.org/hash-functions-and-list-types-of-hash-functions/

Hash Table. (n.d.). Programiz. https://www.programiz.com/dsa/hash-table

Implementation of Hashing with Chaining in Python. (2022). GeeksforGeeks. https://www.geeksforgeeks.org/implementation-of-hashing-with-chaining-in-python/